参考书籍：《C++并发编程实践》

问题：

* 常用命令产生的是物理多线程还是实际多线程？

1. C++98(1998)标准不承认线程的存在，只有在C++11（2011）标准下，才能编写不依赖平台（编译器相关扩展，如C语言中流行的多线程API——POSIX标准中的C标准和Microsoft Windows API中的那些，由第三方加在编译器中）扩展的多线程代码。当前的很多C++编译器为多线程编程者提供了对应(平台相关)的API；当然，还有一些与平台无关的C++类库(例如:Boost和ACE)。Boost线程库与C++11线程库相似
2. 并发的两种途径：C++标准并未对进程间通信提供任何原生支持，所以使用多进程的方式实现，这会依赖与平台相关的API。因此，本书只关注使用多线程的并发，并且在此之后所提到“并发”，均假设为多线程来实现。注意区分实体多线程和虚拟多线程。有些核心物理上是有多个线程的。单核多线程的不会出现同时访问一个内存的情况，因为所有线程也相当于串行工作。因而，一条指令的操作不会带来数据共享问题，多条指令的更改可能造成共享数据问题。可能两个线程都正好要更改某个数据。

* 进程：在2个核心中分别运行2个任务，这是硬件线程。独立的进程可以通过进程间常规的通信渠道传递讯息(信号、套接字、文件、管道等等)。任务并行
* 线程：在2个核心通过任务切换运行2个任务。线程很像轻量级的进程：每个线程相互独立运行，且线程可以在不同的指令序列中运行。但是，进程中的所有线程都共享地址空间，并且所有线程访问到大部分数据——全局变量仍然是全局的，指针、对象的引用或数据可以在线程之间传递。共享内存的灵活性是有代价的：如果数据要被多个线程访问，那么程序员必须确保每个线程所访问到的数据是一致的。数据并行。

1. 使用并发主要原因有两个：关注点分离(SOC)和性能。

* 分离：通过将相关的代码与无关的代码分离，可以使程序更容易理解和测试，从而减少出错的可能性
* 性能：多进程减少总运行时间。多线程使一个循环时间长的不至于影响时间短的。

1. 知道何时不使用并发与知道何时使用它一样重要。基本上，不使用并发的唯一原因就是，收

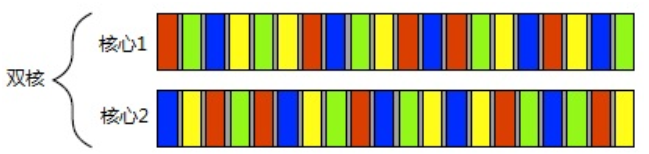
益比不上成本。使用并发的代码在很多情况下难以理解，因此编写和维护的多线程代码就会

产生直接的脑力成本，同时额外的复杂性也可能引起更多的错误。除非潜在的性能增益足够

大或关注点分离地足够清晰，能抵消所需的额外的开发时间以及与维护多线程代码相关的额

外成本(代码正确的前提下)；否则，别用并发。

1. 自己总结

* 多线程虽然也是在一个核心上通过任务切换处理，但是它能不停在不同任务之间切换，不至于让时间长的线程影响时间短的任务。如下图，可看成无论任务复杂度怎样，分配给时间块是相同的。有点像中断，但这是定期的中断。几个线程之间不确定会在哪行代码被打断。所以可能出现某个多操作数据更新任务还没执行完，就被打断了，然后另一个线程正好使用未更新完的数据。这过程类似于几个线程并行。
* 芯片制造商越来越倾向于多核芯片的设计，即在单个芯片上集成2、4、16或更多的处理器，从而获取更好的性能。因为内存是动态的，不同核心不清楚其他核心相应变量的地址。忽略额外的开销，单核n线程的运行总时间是多核单线程的n倍
* 1

1. 线程在声明时自动启动线程，启动后需要明确是要等待线程结束(加入式joined——参见2.1.2节)，还是让其自主运行(分离式detached——参见2.1.3节)。如果不等待线程（detached），就必须保证线程结束之前，可访问的数据得有效性。若线程一定要使用分离式调用局部变量，可以先将数据复制到线程中，而不是使用指针或引用共享数据的形式。
2. 如果想要分离一个线程，可以在线程启动后，直接使用detach()进行分离。如果打算等待对应线程，则需要细心挑选调用join()的位置。
3. 当线程或函数结束时，局部对象就要被逆序销毁了。因此，thread\_guard对象g是第一个被销毁的。拷贝构造函数和拷贝赋值操作被标记为 =delete ③，是为了不让编译器自动生成它们。直接对一个对象进行拷贝或赋值是危险的，因为这可能会弄丢已经加入的线程。通过删除声明，任何尝试给thread\_guard对象赋值的操作都会引发一个编译错误。P27

thread\_guard函数设计和lock\_guard函数有点相似，利用了构造和析构函数，都是RALL语法

1. C++标准库中有很多资源占有(resource-owning)类型，比如 std::ifstream , std::unique\_ptr 还有 std::thread 都是可移动(movable)，但不可拷贝(cpoyable)。
2. 线程间潜在问题就是修改共享数据，致使不变量遭到破坏。不变量（如链表）修改完成后才能被其他线程调用，有时完成一次修改需要较长时间，这时可能会在不同线程之间切换。
3. 多线程数据共享面临的问题

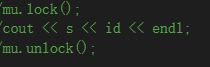
* 条件竞争：最常见的错误。在不变量（如链表）遭到破坏时（更新未完成）访问它，会造成意想不到的错误。恶性条件竞争通常发生于完成对多于一个的数据块的修改时，例如，对两个连接指针的修改。因为出现的概率太低，条件竞争很难查找，也很难复现。每次运行情况都不太一样。条件竞争通常是时间敏感的，所以程序以调试模式运行时，它们常会完全消失，因为调试模式会影响程序的执行时间

解决方法：

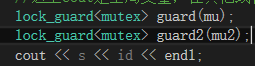
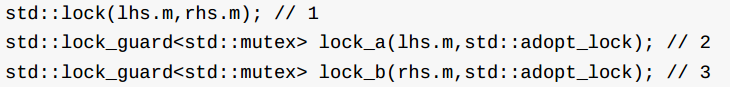
* 最简单的且最常用的办法就是对数据结构采用某种保护机制，确保只有进行修改的线程才能看到不变量被破坏时的中间状态。从其他访问线程的角度来看，修改不是已经完成了，就是还没开始。
* 另一个选择是对数据结构和不变量的设计进行修改，修改完的结构必须能完成一系列不可分割的变化，也就是保证每个不变量保持稳定的状态，这就是所谓的无锁编程(lock-free programming)。不过，这种方式很难得到正确的结果。
* 使用事务(transacting)的方式去处理数据结构的更新(这里的"处理"就如同对数据库进行更新一样)。所需的一些数据和读取都存储在事务日志中，然后将之前的操作合为一步，再进行提交。当数据结构被另一个线程修改后，或处理已经重启的情况下，提交就会无法进行，这称作为“软件事务内存”(software transactional memory

(STM))。理论研究中，这是一个很热门的研究领域。这个概念将不会在本书中再进行介绍，因为在C++中没有对STM进行直接支持。

1. 使用互斥量保护共享数据（P43，内容较多，看文章）：互斥量（当定义的锁在一个地方被锁住时，另一个地方会等待解锁再继续执行）当访问共享数据前，使用互斥量将相关数据锁住，再当访问结束后，再将数据解锁。C++中通过实例化 std::mutex 创建互斥量，通过调用成员函数lock()进行上锁，unlock()进行解锁。C++标准库为互斥量提供了一个RAII语法的模板类 std::lack\_guard ，其会在构造的时候提供已锁的互斥量，并在析构的时候进行解锁，从而保证了一个已锁的互斥量总是会被正确的解锁。一个迷失的指针或引用，将会让这种保护形同虚设。不过，检查迷失指针或引用是很容易的（可以将变量作为类private，只在public函数操作并加锁，且不能将变量以指针或引用暴露给外部改变），只要没有成员函数通过返回值或者输出参数的形式向其调用者返回指向受保护数据的指针或引用，数据就是安全的。虽然某些情况下，使用全局变量没问题，但在大多数情况下，互斥量通常会与保护的数据放在同一个类中，而不是定义成全局变量。互斥量和要保护的数据，在类中都需要定义为private成员，这会让访问数据的代码变的清晰，并且容易看出在什么时候对互斥量上锁。当所有成员函数都会在调用时对数据上锁，结束时对数据解锁，那么就保证了数据访问时不变量不被破坏
2. 死锁，它的最大问题就是由两个或两个以上的互斥量来锁定一个操作。避免死锁的一般建议，就是让两个互斥量总以相同的顺序上锁：总在互斥量B之前锁住互斥量A，就永远不会死锁。但这种方法也会存在问题。幸运的是，C++标准库有办法解决这个问题， std::lock ——可以一次性锁住多个(两个以上)的互斥量，并且没有副作用(死锁风险)。多次加锁一般是因为调用外部函数时，在函数中又包含了一些锁。例子只是极端情况，可能两个锁离得再远一点
3. 加锁的几种方式：加锁并不是直接保护数据，而是，不同线程会先获取锁的状态，锁住时会等待解锁再继续运行，这样，如果数据通过引用或指针在其他地方被更改也是有可能的。注意mu相当于一个全局mutex实例，而guard2是一个局部lock\_guard实例，不断在构造和析构

* 函数如下，这种情况可能因为lock后异常，导致lock死锁。mu称为互斥量
* 利用lock\_guard能够解决上面的问题，函数结束或者异常出现lock\_guard都会被析构，会在析构函数中解锁mu2。一般用{}来设定lock\_guard作用域

1. 避免死锁的进阶训练

* 避免嵌套锁
* 避免在持有锁时调用用户提供的代码
* 使用固定顺序获取锁
* 养成按顺序加锁的习惯
* 使用lock加锁，lock能够管理加锁的顺序，std::adopt\_lock 参数除了表示 std::lock\_guard 对象已经上锁外，还表示现成的锁，而非尝试创建新的锁。
* 使用锁的层次结构：不是C++标准的一部分，但是它写起来很容易

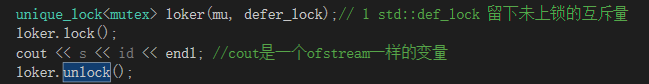
1. 互斥量位置两者会相等待。std::unqiue\_lock 使用更为自由的不变量，这样 std::unique\_lock 实例不会总与互斥量的数据类型相关，使用起来要比 std:lock\_guard 更加灵活。std::unique\_lock 会占用比较多的空间，并且比 std::lock\_guard 稍慢一些。这是保证灵活性要付出的代价。另外其还可通过调用 std::move() 来执行移动所有权操作。

它有多种加锁方式：

* 这个用法和lock\_guard相似

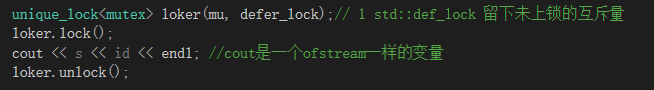
但这样声明相当于未上锁，注意defer\_lock和adopt\_lock的用法

* 这个用法相对把mu赋值给了loker，loker异常时能析构

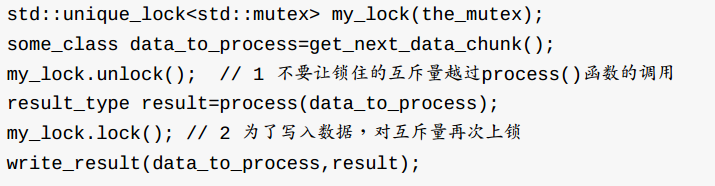


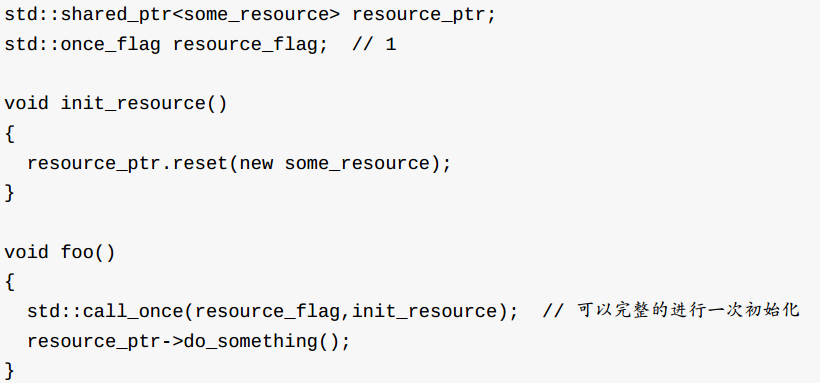
1. 可以使用两种不同的方式给互斥量在不同地方上锁

线程1

线程2

1. 为了使锁的粒度细一些可以如下操作，这就是unique\_lock的灵活性的用处，它还可以多次加锁解锁。注意它是变量有自己的作用域。要考虑加解锁的过程是否能换来较大性能提升



1. 延迟初始化(Lazy initialization)：使用 std::once\_flag 和 std::call\_once 比显式使用互斥量消耗的资源更少，特别是当初始化完成后。
2. 1